NLP: Computational Semantics

Dan Garrette
dhg@cs.utexas.edu

December 27, 2013

1 First-Order Logic

- $P \wedge Q$: P and Q are both true
- $P \vee Q$: Either P is true or Q is true or both are true
- $P \Rightarrow Q$: If P is true, then Q must also be true
- $\neg P$: P is false
- $\forall x.P(x)$: For all x, $P(x)$ is true
- $\exists x.P(x)$: There exists an x such that $P(x)$ is true

2 Lambda Calculus

- $\lambda x.f(x)$: An anonymous function that takes x as an argument and returns $f(x)$
- β-reduction: When the function $\lambda x.f(x)$ is applied some value y, we insert y for each occurrence of x. So, $(\lambda x.f(x))(y)$ becomes $f(y)$.
- α-conversion: Since the x in $\lambda x.f(x)$ is just the name of a parameter, we can replace it with any other parameter name. So $\lambda x.f(x)$ can be rewritten as $\lambda y.f(y)$.

3 Semantics

- Richard Montague
- Principle of Compositionality: the meaning of an expression is the the combination of the meanings of its parts.

Proper nouns are "individuals".

$[\![\textbf{Mary}]\!]$: $mary'$

Intransitive verbs are functions over "individuals", saying that the individual does that thing. Individuals are always represented with a lower-case variable. Functions over individuals are called "properties"; they indicate that the individual "has that property".

$[\![\textbf{sleeps}]\!]$: $\lambda x.sleeps'(x)$

Example: "Mary sleeps"

$$[\![\textbf{Mary sleeps}]\!]: [\![\textbf{sleeps}]\!]([\![\textbf{Mary}]\!]) = (\lambda x.sleeps'(x))(mary') = sleeps'(mary')$$

So *Mary* has the property of *sleeping*.

Transitive verbs are functions over individuals that produce a new function over individuals. So a transitive verb takes a first individual and produces a new function that looks something like an intransitive verb.

$[\![\textbf{sees}]\!]$: $\lambda x.\lambda y.sees'(y, x)$

Example: "Mary sees Bill"

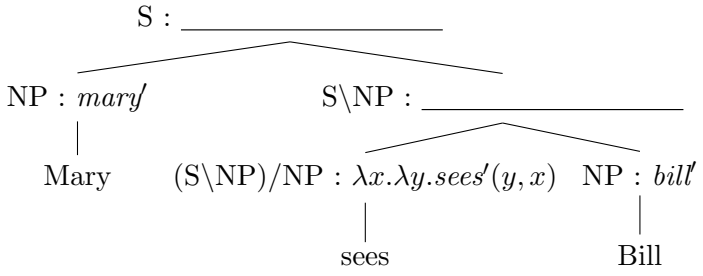
$$\begin{aligned}[\![\textbf{Mary sees Bill}]\!]: \ & ([\![\textbf{sees}]\!]([\![\textbf{Bill}]\!]))([\![\textbf{Mary}]\!]) \\ & ((\lambda x.\lambda y.sees'(y, x))(bill'))([\![\textbf{Mary}]\!]) \\ & (\lambda y.sees'(y, bill'))(mary') \\ & sees'(mary', bill')\end{aligned}$$

We can see that *sees* applies first to the direct object *Bill* to produce a property that looks like an intransitive verb *sees_Bill*. This new function applies to the subject *Mary* to indicate that Mary has the property of seeing Bill.

4 Syntax-driven semantics (with CCG)

Up until this point, we have seen that each word in a sentence can have an associated meaning representation, and that those meaning can be composed to get the meanings of larger phrases, but we have not given any indication of how we know the order in which these pieces should be composed. One natural way is to use syntax to drive this process since, after all, the syntax of a sentence determines how we read its meaning.

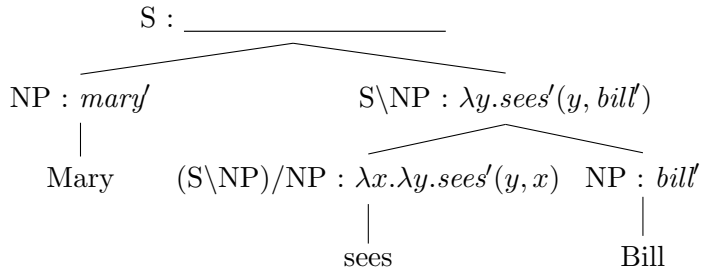
In CCG, every token has an associated CCG category. We can extend this to use semantics by attaching the meaning of the token to its category.



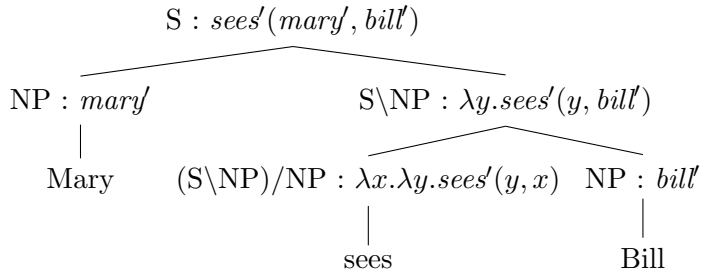
Now we can use the category combinations to drive the semantics combinations.

The tree tells us that *sees* combines first with *Bill*, and that *sees* is the function and *Bill* is the argument (since the category of *sees* is (S\NP)/NP, which takes an NP argument to its right, which

is exactly what *Bill* is). So just as the category of *sees* applies to the category of *Bill* to produce a combined category (S \NP), the semantics of *sees* applies to the semantics of *Bill* to produce a combined semantics:



This process continues for the combination of *Mary* and *sees Bill*.



5 Quantification

Nouns are functions over individuals that say that that individual has the property of being that noun.

⟦**dog**⟧: $\lambda x.dog'(x)$

Quantifiers are functions over two properties. Property variables are written with upper-case letters.

The semantics of a **universal quantifier** like *every* takes two properties and says that every individual that has the first property must also have the second property:

⟦**every**⟧: $\lambda P.\lambda Q.\forall x.[P(x) \Rightarrow Q(x)]$

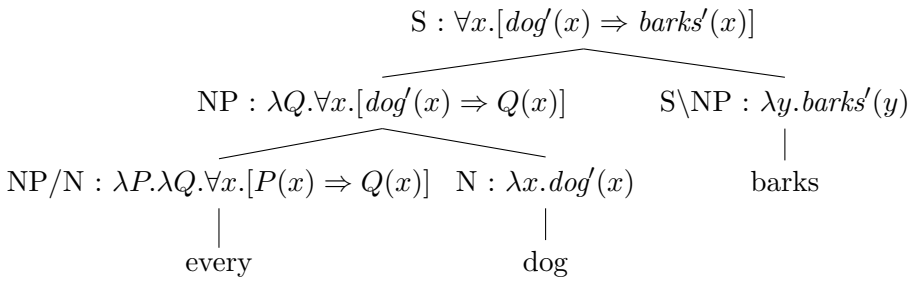
The semantics of an **existential quantifier** like *a* takes two properties and says that there is some individual that has both the first property and the second property:

⟦**a**⟧: $\lambda P.\lambda Q.\exists x.[P(x) \wedge Q(x)]$

Example: "every dog barks"

$$
\begin{aligned}
\llbracket \textbf{every dog barks} \rrbracket : \; & (\llbracket \textbf{every} \rrbracket(\llbracket \textbf{dog} \rrbracket))(\llbracket \textbf{barks} \rrbracket) \\
& (\lambda P.\lambda Q.\forall x.[P(x) \Rightarrow Q(x)])(\lambda z.dog'(z))(\llbracket \textbf{barks} \rrbracket) \\
& (\lambda Q.\forall x.[(\lambda z.dog'(z))(x) \Rightarrow Q(x)])(\llbracket \textbf{barks} \rrbracket) \\
& (\lambda Q.\forall x.[dog'(x) \Rightarrow Q(x)])(\llbracket \textbf{barks} \rrbracket) \\
& (\lambda Q.\forall x.[dog'(x) \Rightarrow Q(x)])(\lambda y.barks'(y)) \\
& \forall x.[dog'(x) \Rightarrow (\lambda y.barks'(y))(x)] \\
& \forall x.[dog'(x) \Rightarrow barks'(x)]
\end{aligned}
$$

And with the syntax driving the derivation:



6 Semantic Ambiguity

Semantic ambiguity is possible even when the syntax is unambiguous. For example: "every student took a test" has exactly one syntacic parse, but it has two semantic interpretations:

$[\![\textbf{every}]\!]([\![\textbf{student}]\!], [\![\textbf{a}]\!]([\![\textbf{test}]\!], [\![\textbf{took}]\!])): \forall s.[student'(s) \Rightarrow \exists t.[test(t) \wedge took(s,t)]]$

$[\![\textbf{a}]\!]([\![\textbf{test}]\!], [\![\textbf{every}]\!]([\![\textbf{student}]\!], [\![\textbf{took}]\!])): \exists t.[test(t) \wedge \forall s.[student'(s) \Rightarrow took(s,t)]]$

Semantic ambiguities are more complex to handle, but several frameworks exist to manage them. Generally these use **underspecified** semantic representations that capture some of the semantic structure while leaving some portions incomplete (though contrained to still be legal representations).

One underspecification framework is **Hole Semantics** developed by Johan Bos. The basic idea is that you have "labels" on logical expression, "holes" in the expression, and constriants on which holes can be filled by which labels:

$l_1 : \forall s.[student(s) \Rightarrow h_1]$

$l_2 : \exists t.[test(s) \Rightarrow h_2]$

$l_3 : \lambda x.\lambda y.took(x,y)$

$l_1 \leq h_0$

$l_2 \leq h_0$

$l_3 \leq h_1$

$l_3 \leq h_2$

7 Neo-Davidsonian Semantics

In our current view of logical forms, we may see representations like this:

$[\![\textbf{Mary helped}]\!]: help'(mary')$

[[**Mary helped Bill**]]: $help'(mary', bill')$

[[**Mary helped Bill with homework**]]: $help'(mary', bill', homework')$

[[**Mary helped with homework**]]: $help'(mary', homework')$

But this scheme is problematic because it is inflexible and doesn't capture the generalities among these statements. In this scheme the logical form of *Mary helped Bill with homework* does not entail the logical form of *Mary helped Bill*.

Neo-Davidsonian semantics tries to fix this problem by separating out all of the verb's arguments. It does this by shifting the focus to a discussion about **events** and makes statements about those events. So now the statement "Mary helped" is given a representation that states that there is some event e that is a "helping" event and that Mary is doing the helping:

[[**Mary helped**]]: $\exists e.[help'(e) \wedge \text{agent}(e, mary')]$

Therefore, all other arguments to the verb are simply additional conjuncts:

[[**Mary helped Bill**]]: $\exists e.[help'(e) \wedge \text{agent}(e, mary') \wedge \text{patient}(e, bill')]$

[[**Mary helped Bill with homework**]]: $\exists e.[help'(e) \wedge \text{agent}(e, mary') \wedge \text{patient}(e, bill') \wedge \text{theme}(e, homework')]$

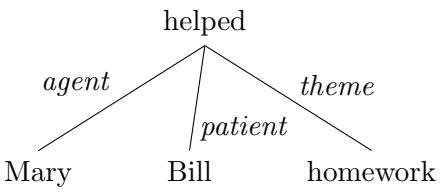
[[**Mary helped with homework**]]: $\exists e.[help'(e) \wedge \text{agent}(e, mary') \wedge \text{theme}(e, homework')]$

And, according to the rules of first order logic, the representation of *Mary helped Bill with homework* entails the representation of *Mary helped Bill*:

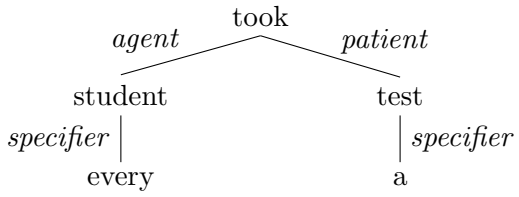
$\exists e.[help'(e) \wedge \text{agent}(e, mary') \wedge \text{patient}(e, bill') \wedge \text{theme}(e, homework')] \vDash \exists e.[help'(e) \wedge \text{agent}(e, mary') \wedge \text{patient}(e, bill')]$

7.0.1 Neo-Davidsonian Semantics and Dependency Grammar

Using (labeled) dependency grammars makes generating Neo-Davidsonian representations straightforward:



(a) Mary helped Bill with homework.



(b) Every student took a test.

8 Discourse Representation Theory (DRT)

DRT is a similar to first-order logic, but it differs in a few key ways. First, expressions in DRT are written in a graphical format known as a Discourse Representation Structure (DRS). Second, DRT

is a *dynamic* logic, meaning that the expressions can be *updated* as more information is available (ie, as more text is read).

A DRS consists of two parts written inside a "box". On top are the *discourse referents*, which are basically existentially quantified variables. Below are the *discourse conditions*, which are logical statements.

Examples:

$$[\![\textbf{fido barks}]\!] = \begin{array}{|c|}\hline \\ \hline \text{barks(fido)} \\ \hline \end{array}$$

$$[\![\textbf{a dog barks}]\!] = \begin{array}{|c|}\hline x \\ \hline \text{dog(x)} \\ \text{barks(x)} \\ \hline \end{array}$$

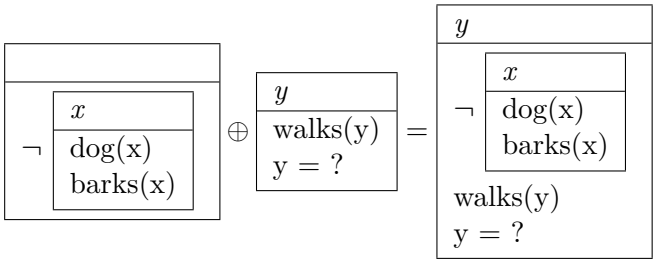
A pronoun would be represented in DRT as a variable whose antecedent must be resolved:

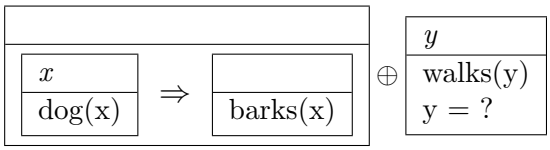
$$[\![\textbf{he walks}]\!] = \begin{array}{|c|}\hline x \\ \hline \text{walks(x)} \\ \text{x = ?} \\ \hline \end{array}$$

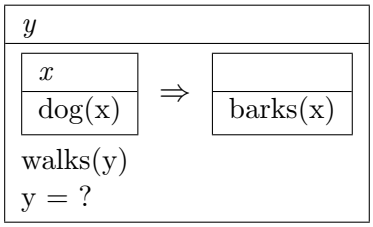
DRT becomes particularly interesting as we add more information (new sentences) to the discourse. When we add a new sentence, we "merge" it into the existing discourse by adding its referents to the existing set of referents and adding its conditions to the existing set of conditions. Once the new sentence is merged in, any unbound pronouns *in the new sentence* can be resolved.

$$[\![\textbf{A dog barks. He walks.}]\!] = \begin{array}{|c|}\hline x \\ \hline \text{dog(x)} \\ \text{barks(x)} \\ \hline \end{array} \oplus \begin{array}{|c|}\hline y \\ \hline \text{walks(y)} \\ \text{y = ?} \\ \hline \end{array} = \begin{array}{|c|}\hline x\ y \\ \hline \text{dog(x)} \\ \text{barks(x)} \\ \text{walks(y)} \\ \text{y = x} \\ \hline \end{array}$$

Note, however, that pronouns can only be resolved to "accessible" referents, which are referents in the current box or any out-scoping box. Thus, the pronoun in this discourse fails to resolve:

$[\![$**No dog barks. He walks.**$]\!] =$ 

$[\![$**Every dog barks. He walks.**$]\!] =$ 

$=$ 

9 Textual Entailment

Typically, when we are working in semantics, we don't merely want to represent meanings, we usually want to use those meanings to perform tasks.

One such task is *textual entailment*, which is determining whether one piece of text entails another. Converting to logical form allows us to use standard first-order theorem proving to determine whether one text entails another:

"Fido barks and sleeps." $\models$ "Fido barks."
$barks'(fido') \wedge sleeps'(fido') \models barks'(fido')$

"Fido barks." $\not\models$ "Fido barks and sleeps."
$barks'(fido') \not\models barks'(fido') \wedge sleeps'(fido')$

"Every dog barks." $\models$ "A dog barks."
$\forall x.[dog'(x) \Rightarrow barks'(x)] \models \exists x.[dog'(x) \wedge barks'(x)]$

"A dog barks." $\models$ "Every dog barks."
$\exists x.[dog'(x) \wedge barks'(x)] \models \forall x.[dog'(x) \Rightarrow barks'(x)]$

"Every dog barks." $\models$ "Fido is a dog and fido barks."
$\forall x.[dog'(x) \Rightarrow barks'(x)] \models dog'(fido') \wedge barks'(fido')]$

10 Issues with Logical Semantics

The principle of compositionality doesn't hold for idioms:

"Bill kicked the bucket." $\models ?$"A bucket was kicked."

It doesn't handle word sense:

"The player picked up the bat." $\models ?$"The player picked up an animal."
"The bat flew out of the cave." $\models ?$"A baseball bat few out of a cave."

It doesn't have any notion of likelihood.

- In some contexts it may be ambiguous whether an idiom is being used or not.
- In some contexts two sense may be *possible*, one sense may be more likely than another.